ROM Allocation of Constant Data in a JVM for Embedded Systems

Diplomarbeit im Fach Informatik

vorgelegt von

Simon Kuhnle

geboren am 5. März 1987 in Kaiserslautern

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dipl.-Inf. Christoph Erhardt** **Dipl.-Inf. Isabella Stilkerich** **Prof. Dr.-Ing. W. Schröder-Preikschat**
Beginn der Arbeit:	**12.08.2013**
Abgabe der Arbeit:	**12.02.2014**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 12.02.2014

Überblick

C ist die Programmiersprache, die den Bereich der eingebetteten Systeme dominiert, da sie sehr ressourceneffizient und hardwarenah ist. In sicherheitskritischen Systemen ist Geschwindigkeit nicht das einzige Kriterium. Eine typsichere Programmiersprache mit Sicherheitsüberprüfungen zur Laufzeit, die Fehlzugriffe im Speicher verhindert, wäre von Vorteil. Inbesondere auf Systemen, die keinen Hardware-Speicherschutz besitzen.

KESO ist eine Java Virtual Machine für eingebette Systeme, die C nach Java *Ahead-of-Time* übersetzt und dadurch die Sicherheitsfeatures von Java und die Geschwindkeit von C vereint. Da RAM eine begrenzte Ressource auf eingebetteten Systemen darstellt, bietet es sich an, konstante Daten im ROM zu allokieren. Unglücklicherweise bietet Java keine Möglichkeit Daten als konstant zu markieren.

In dieser Arbeit wurden dem Java-nach-C-Compiler von KESO neue Analysen hinzugefügt, die konstante Daten entdecken. Ausserdem wurde das Backend des Übersetzers so angepasst, dass konstante Daten im ROM plaziert werden können. Das Laufzeitsystem von KESO verfügt desweiteren über eine automatische Speicherbereinigung. Diese musste angepasst werden, da sie Schreibzugriff auf den Objekt-Kopf benötigt, um korrekt zu funktionieren. Dieses Vorgehen scheitert jedoch bei nicht beschreibbaren Objekten, weshalb der Garbage Collector konstante Objekte im ROM ignorieren muss.

Die verschiedenen Analysen und Anpassungen an Backend und Laufzeitsystem wurden evaluiert. Die Ergebnisse unterscheiden sich sehr von Anwendung zu Anwendung und sind stark abhängig von der Verfügbarkeit von ausnutzbaren Daten. Je nach Anwendung provozierten die Optimierungen keine Veränderung, unter den richtigen Umständen jedoch bis 10% Geschwindigkeitszuwachs.

Abstract

C is the dominating programming language in the field of embedded systems, as it is resource efficient and operating close to the hardware. In safety-critical environments, speed is not the only priority. A type-safe language with runtime security checks, providing memory safety, would be convenient, especially on systems lacking a memory protection unit.

KESO is a Java Virtual Machine targeting embedded systems, compiling Java bytecode to C ahead of time, thus combining the security and safety features of Java and the speed of compiled C code. As RAM is scarce resource on most embedded systems it would be desirable to allocate constant data in the target's read-only memory. Unfortunately, Java does not provide a way declare fields as constant.

In this thesis, new analysis passes were added to KESO's Java-to-C compiler, detecting constant data. Furthermore, functionality was added to the compiler's backend, so the constant data is placed in ROM. KESO's runtime environment features a garbage collector, which had to be adjusted as well, as the garbage collector needs write access to the object headers, in order function properly. This write operation will fail, when it comes to read-only objects, so the garbage collector needs to ignore them.

The different analysis passes and adjustments of the backend and runtime system were evaluated. The results vary from application to application, based on the availability of exploitable data. Depending on the application, the results range from no effect at all to 10% performance increase.

Contents

1 Introduction

Embedded systems are all around us, used for a wide range of tasks, from controlling the thermostat at home, over playing music on MP3 players to supporting brakes in cars. Those systems are tailored for one specific task, equipped with processors of varying complexity, ranging from simple 8-bit AVR microcontrollers to full-blown 32-bit multi-core ARM processors. For performance reasons and restrictions regarding code size and memory usage, software for these systems is written in C most of the time, or even hand-crafted assembly code. Operating so close to the hardware and having only few layers of abstraction is what makes these languages so fast. However, this makes them risky at the same time, creating easy ways to make mistakes, leading to overwritten memory or otherwise unwanted behaviour, especially with microcontrollers that lack memory protection. Using a type-safe programming language, providing compile-time and runtime safety checks, would increase memory safety and remove the necessity of a hardware memory protection unit.

1.1 The KESO Multi-JVM

Java is not the language that usually comes to mind talking about embedded systems, though it is widely used in many other sectors. Its object-oriented programming paradigm, automated memory management, type-safety and various other features make this language so popular. Using the KESO Multi-JVM[10] it is possible to write Java code for statically configured, deeply embedded systems, using the comfort of Java and its security and safety features, like runtime bounds checks. KESO is capable of generating code for different real-time operating system standards like AUTOSAR OS[2] or OSEK[8], and for a variety of processor architectures, including TriCore and AVR. Supporting only statically configured systems and not allowing the loading of additional components at runtime, distinguishes KESO from regular JVMs. This offers many possibilities for optimizations, because it is guaranteed that the whole code base is known at compile time.

Unlike most JVMs, KESO is not a bytecode interpreter or just-in-time compiler, executing native instructions on the CPU. KESO compiles Java bytecode to C *ahead of time*. Compiling the emitted C code with a C compiler of the developer's choice, induces another round of optimizations. The existence of the C compiler disburdens KESO from having to know everything about the target

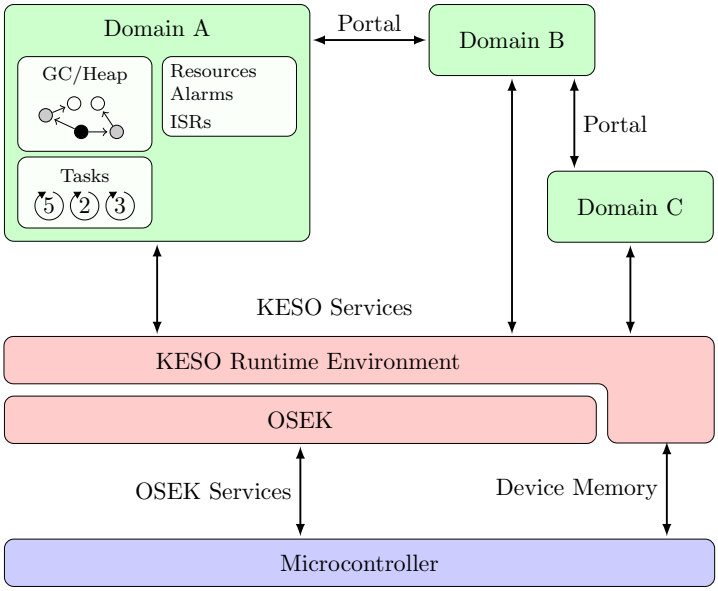

Figure 1.1: Overview of the KESO environment.

architecture and at the same time makes it easy to port KESO to new platforms.

Figure 1.1 gives an overview of the KESO environment. Different tasks are confined to their *domain*, representing their runtime environment. Each domain has its own heap, garbage collection mechanism and other resources, making a domain a JVM instance of its own, thus the term *Multi-JVM*. It is possible to communicate with other domains via KESO's *portals*, an RPC-like mechanism. Other ways of accessing resources or objects of other domains are prohibited, protecting the isolation property of the domains.

1.2 Motivation

Targeting deeply embedded systems, KESO strives to produce code that meets the constraints of that area. KESO already provides a wide range of optimizations, removing dead code, unused methods and fields, and unnecessary runtime checks. On the other hand, read-only data like lookup tables and strings occupy precious RAM space. Reducing the demand for RAM by putting constant data into ROM would reduce the overall cost of the system, since ROM space is significantly cheaper in production cost.

One reason why KESO is not able to put constant data into ROM is that Java has no way to declare fields constant in the first place, like the `const` keyword does in C. The thing that comes close to defining a constant variable is to label a field with the `final` modifier, which forbids the field's reference to change. For primitive data types like `int` this is sufficient, as their "reference" cannot

be changed. However, the content of non-primitive data types, like objects and arrays, is not protected by the `final` modifier. Because of this, KESO has to assume that the content is subject to change and has to allocate space for the array or object in memory.

One thing that is guaranteed to be constant in Java are string literals, which means text in quotation marks. Every string literal used in the code gets placed in the JVM's *constant pool* and will be referenced by its constant pool entry ID in the bytecode. KESO emits string literals in form of C `char` arrays, but is not able to mark it with C's `const` modifier, because of the way the garbage collector works.

The scope of this thesis is to detect different kinds of constant data in the applications compiled by KESO, add functionality to put constant data into ROM, thereby eliminating the runtime efforts for its allocation and initialization, ultimately reducing the application's memory consumption. Furthermore, KESO's runtime environment, the garbage collector in particular, has to be adjusted to handle constant data properly. Some backends require specific changes generating code to place data in ROM and access it as well, because of their architecture's design.

1.3 Outline of this Thesis

The next chapter provides the required background information, giving an overview of KESO's Java-to-C compiler JINO, a detailed insight into the optimization passes relevant to this thesis, as well as some important data structures. Chapter 3 describes the newly introduced compiler passes needed to find constant data and the functionality that was added to the backend and the runtime environment. The evaluation in Chapter 4 measures the impact on binary size and execution performance caused by these optimizations. The thesis then concludes with a summary.

2 Background

This chapter gives a quick overview of how KESO's Java-to-C compiler JINO is organized and describes the already existing compiler passes that will later be used to detect constant data. Additionally, some important data structures and the relevant information about the garbage collector will be outlined.

2.1 The JINO Java-to-C compiler

KESO's heart is the Java-to-C compiler JINO. It's functionality is split into three major components. The first part parses Java bytecode into JINO's intermediate code representation, then optimizations of the intermediate code take place. The last part is responsible for generating the application's C code.

2.1.1 Frontend

The frontend is responsible for parsing the system's configuration files and the preparing the the Java source. It transforms (`.java`) files over to Java bytecode (`.class`) and finally to JINO's own intermediate code representation. JINO uses the regular Java compiler to generate the `.class` files, removing the need for functionality for lexing and parsing the original Java code. JINO then parses the Java bytecode files and extracts all the relevant information for each class. Methods get translated to intermediate code representation, split up into basic blocks, which are composed of instruction nodes.

Listing 2.1 shows the Java bytecode of a recursive method computing the greatest common divisor of two integers and Listing 2.2 the respective JINO intermediate code representation. JINO's intermediate code is very similar to JVM bytecode, although opposed to the JVM's stack machine layout, JINO references operands as results of previous instructions.

2.1.2 Intermediate Code Analysis and Transformation

After the intermediate code is created, JINO runs various analysis and transformation passes on it. Starting with gathering of basic information, like class type information and simple constant propagation, the code gets transformed into the static single assignment (SSA) form, which is required by further passes.

```
public static int gcd(int, int);
  Code:
   0: iload_1
   1: ifne  6
   4: iload_0
   5: ireturn
   6: iload_1
   7: iload_0
   8: iload_1
   9: irem
  10: invokestatic  #2; //Method gcd:(II)I
  13: ireturn
```

Listing 2.1: JVM bytecode of a recursive GCD method.

```
gcd(II)I {
_B0: (0/2; LiveIn: [i0, i1]; PhiIn: []; LiveOut: [i0, i1])
     0:    %0 = IReadLocalVariable i1
     1:    %1 = IConstant 0
     1:    %2 = NEConditionalBranch %0, %1, [_B4, _B6]
_B4: (1/1; LiveIn: [i0]; PhiIn: []; LiveOut: [i0])
     5:    %3 = Goto [_B15]
_B6: (1/1; LiveIn: [i0, i1]; PhiIn: []; LiveOut: [i2_1])
     7:    %4 = IReadLocalVariable i0
     8:    %5 = IReadLocalVariable i1
     9:    %6 = IRem %4, %5
    10:    %7 = IStoreLocalVariable i3_0, %6
    10:    %8 = IReadLocalVariable i3_0
     6:    %9 = IReadLocalVariable i1
    10:   %10 = InvokeStatic Example.gcd(II)I %9, %8
    13:   %11 = IStoreLocalVariable i0, %10
    13:   %12 = Goto [_B15]
_B15: (2/0; LiveIn: []; PhiIn: [i0, i2_1]; LiveOut: [])
    15:   %13 = Epilog i0
}
```

Listing 2.2: JINO intermediate code of the recursive GCD method.

Each optimization pass comes with its own list of passes that need to be run before and after the current pass is executed. The passes are executed in an optimization loop, until either a certain number of iterations has passed or no pass reports any changes. The transformation passes aim to generate code that is as lean and fast as possible. They include removal of unused methods and fields, removal of unreachable code, and inlining of methods. The different passes influence each other greatly. For example, a folded constant could transform a conditional branch into an explicit branch, rendering the alternative path unreachable. If this path contained the only invocation of a certain method, this method can now removed. After the optimization loop finishes, the SSA form gets de-constructed and the intermediate code is prepared for the translation process.

2.1.3 Backend

The backend translates the intermediate code to C code, class by class, method by method. JINO uses the previously gathered information about used types, code reachability and content of variables to omit the classes and methods that were marked as unreachable. It inserts crucial runtime safety checks like `null`-pointer and array-bounds checks in all places where the previous analyzes passes did not prove them to be redundant. The backend is responsible for generating everything that is needed to compile and deploy the emitted C code, like Makefiles, OSEK files describing the details of the target system as well as the application. The backend also emits the rest of the runtime environment, which was not translated from Java code, but comes in form of C files, like the garbage collector code for example. Architectures supported by the backend, besides Intel x86, are the Infineon TriCore, which is a popular architecture in the automotive industry, as well as the Atmel AVR, a simple and cheap 8-bit microcontroller, also used in the automotive sector and embedded systems in general.

2.2 Important Data Structures

JINO has to keep track of different types of information, mostly related to Java being an object-oriented programming language. For instance, to ensure one of the central features of Java, type safety, JINO has to have global knowledge about the class hierarchy and the different object types. Other information, like the size of a class is also important, so the memory allocator knows how much space is needed in case an object is allocated at runtime. What follows are the descriptions of the most important data structures of JINO.

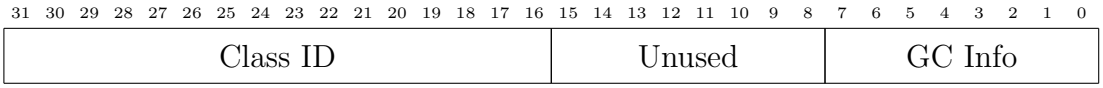

Figure 2.1: Layout of the object header, for a 32-bit system.

2.2.1 Class Storage

Every class used in the application is represented by an entry in JINO's global class storage. The class storage is composed of entries containing the following information about each class:

- Size of the object
- Number of interfaces implemented by the class
- Number of object references

The index of each class in the class storage also serves as its class identifier.

2.2.2 Virtual Method Table

Another OOP-specific data structure JINO needs to maintain is the virtual method table. If a sub-class overrides a method of its base class, there are two methods with the same method signature. Both methods are saved in the virtual method table, together with the class ID of their respective class. When the compiler comes across a invocation of this method, it has to emit a special virtual method call instruction, so the correct method is called at runtime. The virtual call instruction then uses the class ID from the object's header to lookup the correct method to be called.

2.2.3 Object and Array Header

Additionally to the content of the object itself, every object needs a header, containing the object's class identifier and one byte reserved for the garbage collector. This garbage collector byte is used to register whether the object is still referenced or if its memory can be freed. Array headers contain the same information as the regular object header, but have an additional entry representing the size the of the array. Figure 2.1 shows the layout of an object header layout for 32-bit systems, with a 16-bit class ID, and one byte reserved for garbage collector information. The rest of the 32-bit header is currently not in use.

2.3 KESO's Garbage Collector

Like every Java virtual machine, KESO comes with automatic memory management, taking care of the removal of unused objects, so the developer does not have to. KESO includes two different variants of garbage collection, configurable for every domain. One variant works during the application's idle time, the other one periodically stops the application.

As mentioned above, every object has an object header, in which one byte is reserved for garbage collector information. Whenever the garbage collector searches for objects that can be freed, it starts looking at the references in its *root set*. The root set consists of static class fields and objects on the current stack. Starting from the root set, the garbage collector visits the respective object references of every object, setting a bit in a bitmap representing all objects that are still reachable from the root set. To prevent infinite loops while following references, each visited object is marked with a specific color, represented by a bit in the object header. From the color bit, the garbage collector knows that it already visited this object. This color changes every time the garbage collector runs, so every visited object has to be colored with the color of the current run. This is done by toggling one bit in the object's header. At the end of the garbage collector run, objects gets freed, if their respective bit in the bitmap has not been set.

2.4 Data Flow Analysis

The main advantages of Java, like object-oriented programming, and runtime security checks, come at a price. The need to resolve virtual method invocations, and performing array bounds checks, etc. affects the code size and execution performance. To keep their impact as low as possible, JINO incorporates a powerful data flow analysis pass described in [4], based on the *Sparse Conditional Constant* algorithm by Wegman and Zadeck [11]. The pass analyzes the intermediate code in a control-flow sensitive manner, visiting the intermediate code in the order in which the program code would normally be executed. That way, it computes information about object types, reachability of basic blocks and methods, usage of classes and content of fields and stack slots. This information can later be used to remove some array bounds check, in case the data flow analysis has proven that the index in use is always within the boundaries of the array's size. Furthermore, the gathered object type information can remove virtual method table lookups, if the analysis shows that there is only one possible method candidate that can be called.

2.4.1 Information Representation

The algorithm computes *lattice values* for every variable it passes on its way through the basic blocks. This lattice value represents the gathered knowledge about the type and content of the variable. Each node consists of lattice cells, which can be described as a container for the lattice elements of the operands and for the result of the node itself.

A lattice value can be in one of the following three states:

- No data flow information available yet (default preset)
- Constant value
- Non-constant value (no further exploitation possible)

It is important to note that a lattice's state can only be "lowered", so only going from the "no information available" state to constant or non-constant is possible, but there is no way to escape the "non-constant value" state.

If the data flow analysis evaluates a binary expression, like for instance an addition, it does so by inspecting the cells of both operands. Using the information for both values, gathered from previous assignment statements for example, the data flow analysis performs the operation, for example the addition, saving the result in the expression node's cell. In the case that both operand cells are constant, the result's cell will be constant as well. If one or both cells are in the non-constant state, the result of the expression is non-constant as well, putting the expression's cell in that state, too.

The "constant" state of the lattice can be defined in a more fine-grained manner for specific types of cells. For example, an integer variable normally gets assigned more than one value over the course of a program. So the lattice of an integer variable in JINO can therefore go from containing one value to a set of values. After a certain threshold of members of the set is reached, the information precision is reduced to a value range. The information about a possible range of values can still be useful, when checking if an index is within array boundaries for example. Lattices of object references can also be constant, or valid, non-`null` objects, or unknown. The backend uses this information to omit null-pointer checks, if the analysis can prove that the object reference is always valid.

2.4.2 Propagation of Information

The data flow analysis algorithm starts analyzing the basic blocks of all possible entry points of the program, like the main method, but also interrupt handlers and the like. Further basic blocks are analyzed, if it becomes clear that these blocks will be visited, by their respective branch instructions. The algorithm evaluates conditional branches based on the gathered cell information. If the lattice cell information shows that a condition can or cannot be met, it will only visit the

basic block that would be taken depending on the outcome of the conditional statement.

If the lattice value of a variable changes due to an assignment statement, all nodes that depend on that lattice information need to be revisited, to update the cells containing the lattice value of the variable. For example, if a node writes to a certain field, all other nodes using this field need to be revisited, to propagate the new information.

The algorithm terminates when no more nodes need to be visited, as all information was propagated and every lattice value has reached a stable state.

2.4.3 Usage of the Gathered Information

The results of the data flow analysis are used in a variety of other passes. In the whole program optimization pass, basic blocks that were never visited during the data flow analysis will be removed. With the help of the computed object type information, the number of possible method candidates in a virtual method call can also be reduced. The virtual method invocation could even be converted to a regular invocation, if only one possible candidate is left.

2.5 Extended Escape Analysis

One of Java's biggest assets is its automatic memory management. There is no need to free memory explicitly, as the garbage collector will take care of objects that are not reachable anymore and frees the heap space they claim. Some objects even last only as long as the method they were created in. Instead of iterating over lists of available memory slots, trying to find heap space with the appropriate size for these objects, they could be allocated cheaply on the stack, if their size permits it. Allocating stack memory normally just requires a certain value to be subtracted from the stack base register, and freeing the memory happens implicitly in the epilogue of the function.

The extended escape analysis (EEA) described in [6], based on the algorithm developed in [3], detects if objects *escape* their method. Escaping means that they are being referenced outside of the method they were created in. Objects that do not escape their method can be allocated on the stack instead of the heap. This speeds up both the allocation and the deallocation process and also reduces the workload of the garbage collector. In order to determine if objects escape, the EEA pass performs an extensive alias analysis, keeping track of assignment statements and references used as method parameters and return values.

The relevant information for this thesis gathered from the EEA is alias information, which is needed to see which fields and stack slots point to arrays. That information is used to correctly propagate lattice information of arrays in the data flow analysis.

2.5.1 The Connection Graph

The algorithm builds a *connection graph* (CG) for each method. In this graph, vertices representing field references and stack slots point to vertices representing allocated objects. There are *object nodes*, representing allocated objects. Field reference nodes represent fields, pointing to object nodes with field reference edges. Global reference nodes are `static` class members. Stack slots are represented by local reference nodes. Method parameters and return values are called *actual* reference nodes.

The EEA starts by analyzing the intermediate code on a per-method basis. Each object allocation, that means an invocation of `new`, in the code creates a new object node. Method calls create nodes for the respective parameters and the objects they are representing. Assignment statements create a reference node (if none exists yet) for the left hand side and an edge from the variable holding the reference to the object. If no object node for the specific object is available yet, as it is allocated in another method, a *phantom node* is created, to be resolved later in the interprocedural analysis.

Every object has a default escape state, set to *local*, which means stack allocatable. The escape state changes accordingly, if new references point to the object. For example, if a static field reference is attached to an object node, this object escapes globally.

After the intraprocedural analysis is finished, the EEA propagates each callee's information to its callers, resolving the phantom nodes and spreading the escape state information. The details of the interprocedural analysis can be found in the thesis of Clemens Lang [6] and will not be further discussed here.

2.5.2 Example

Listing 2.3 shows a simple example of two arrays, a two dimensional static field and a regular one dimensional field. The generated C code is shown in Listing 2.4 and corresponding connection graph is shown in Figure 2.2. *oned* is as a member of *obj0*, which is always the `this` object representing the class instance. *twod* is a global reference node, connected to a object node representing the first dimension of the array. Two field array edges originate from that object node, each connecting to nodes with the corresponding array index of the two dimensional array. All stack slots used in the array writes and reads are pointing to their respective object nodes representing the allocated array.

2.6 Summary

KESO's Java-to-C compiler JINO compiles Java code *ahead of time* to C code, focusing on generating compact code suitable for embedded systems, which is supported by a wide range of compiler optimization passes. Two of those passes were

```
1  public class Example implements Runnable {
2
3          static private byte[][] twod = { {0}, {1} };
4          private byte[] oned = { 23 };
5
6          public void run() {
7                  oned[0] = 42;
8                  System.exit(twod[0][0]);
9          }
10 }
```

Listing 2.3: Example Java code for connection graph example.

```
1  void c7_Example_m4_run(object_pointer obj0)
2  {
3          jint i2_0;
4          object_pointer obj1_0;
5          object_pointer obj1_1;
6          object_pointer obj1_2;
7
8          obj1_0 = (ACCFIELD_C7_EXAMPLE_C7F2_ONED(obj0));
9
10         BYTE_ARRAY_LEA(obj1_0, (0)) = 0x2a;
11         obj1_1 = SC7_EXAMPLE_C7F1_TWOD(((domain_t *) &dom1_DDesc));
12
13         obj1_2 = OBJECT_ARRAY_ALOAD(obj1_1, (0));
14         i2_0 = BYTE_ARRAY_ALOAD(obj1_2, (0));
15
16         ShutdownOS((StatusType) i2_0);
17
18         return;
19 }
```

Listing 2.4: Generated C code from Java code in listing 2.3.

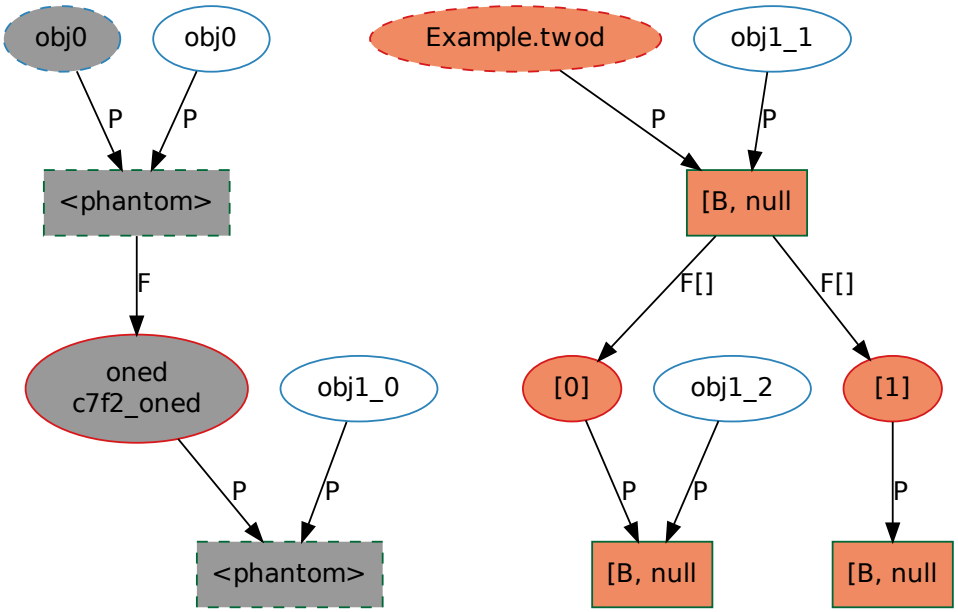

Figure 2.2: Simple connection graph generated by the EEA.

described in detail, as they gather information needed in the following chapter, in which this information will be used in the detection of constant data. The data flow analysis gathers information about field content, and the extended escape analysis collects alias information. Furthermore, some of KESO's data structures and details of garbage collector internals, especially the coloring of objects, were described, as they are subject to changes in the realization of ROM allocated data support in KESO.

3 Design and Implementation

This chapter presents how the goal of this thesis—detecting constant data and placing it in ROM—is achieved, by adding compiler passes, extending existing ones, as well as adjusting the backend and runtime environment.

Java does not offer a way to declare fields as constant, in a way that it is guaranteed that their content cannot be changed at runtime. KESO needs a way to identify if a field's content does not change after its initialization, before being able to place it in read-only memory. Additionally, the KESO runtime environment prevents the ROM allocation of constant objects.

The chapter describes how constant arrays are found, by tracking their content with the already existing data flow analysis. Accuracy of the array's content is ensured by retrieving aliasing information for the arrays. For a field array to be constant, its reference must not change during runtime. This is ensured by another analysis pass, checking if non-`final` fields fulfill the requirements of the `final` modifier. JINO's backend will be taught how to emit the content of the constant data in way it is placed in read-only memory of the target architecture. Finally, the garbage collector is adjusted, so it ignores constant objects, preventing it from trying to write to their headers.

3.1 Constant Arrays

In application software, there are many different uses for constant arrays. They can, for example, serve as lookup tables for pre-calculated sine values, state transition tables of parsers, or substitution boxes in cryptography. However, Java does not provide a way to declare array content immutable, so JINO cannot tell if an array is constant and will treat it like any other array, allocating space and writing its initial values to it. Code like this initialization of a `final int` array:

```
private static final int[] foo = {1,2,3};
```

translates to Java bytecode shown in Listing 3.1, allocating space for the array and initializing every array index. Based on this bytecode, JINO emits the C code shown in Listing 3.2. It would be more efficient to find out if an array like *foo* is constant, and make it possible to emit it fully initialized in the C code. Instead of allocating memory and wasting runtime by initializing the allocated array, like it is done in Listing 3.2, a reference to the initialized array could substitute the allocation.

```
1  iconst_3
2  newarray int
3  dup
4  iconst_0
5  iconst_1
6  iastore
7  dup
8  iconst_1
9  iconst_2
10 iastore
11 dup
12 iconst_2
13 iconst_3
14 iastore
15 putstatic #3; //Field foo
```

Listing 3.1: JVM bytecode of an array initialization.

```
1          obj1_0 = keso_alloc_int_array((3));
2                   INT_ARRAY_LEA(obj1_0, (0)) = (1);
3                   INT_ARRAY_LEA(obj1_0, (1)) = (2);
4                   INT_ARRAY_LEA(obj1_0, (2)) = (3);
```

Listing 3.2: C code emitted by JINO initializing an array.

The following will describe how constant arrays can be found and how JINO emits them as initialized structures.

3.1.1 Gathering Data Flow Information

To gather information about constant array content, support for tracking array information was added to the data flow analysis. As described in Chapter 2.4, the data flow analysis visits the code in a control-flow sensitive manner, calculating *lattice* information for every instruction node in every basic block that was declared reachable based on evaluated conditional branches.

The array specific part is pictured in a simplified manner in Figure 3.1 and works as follows. If the analysis comes across an array allocation, it creates an empty array cell object, containing information about the requested array size and content type. The array cell also contains an actual array of the requested size, used for saving the cells of the operands written to the array. If the lattice cell of the operand representing the requested array size is not a constant, the array cell's array will be set to `null` and every future read or write to this array will yield in a lattice state, representing the "unknown" state.

On an array write, three lattice cells need to be examined: the array cell, the index, and the operand. The operand cell gets written to the array inside the

array cell object. The index can either be a constant integer, in which case the operand cell gets written to that exact index. If the index cell is in the non-constant state, we *merge* the content of the operand cell with every cell of the array cell array, because the destination is not known, so it could be written to any index of the array.

When reading from an array the data flow analysis takes the cell information for the array and the index and returns the cell that is saved in the array cell's array, from the position of the index cell. So this is either cell at the position of the index, or a "meet" over all cells in that array, in case the index is not constant. This might still be valuable information, for example, when reading from integer array it can return a cell containing the range of values in that array. In case that cell is assigned to an integer used as an index for another array access and the range of the cell is within the boundaries of the array to be accessed, the bounds check can be omitted.

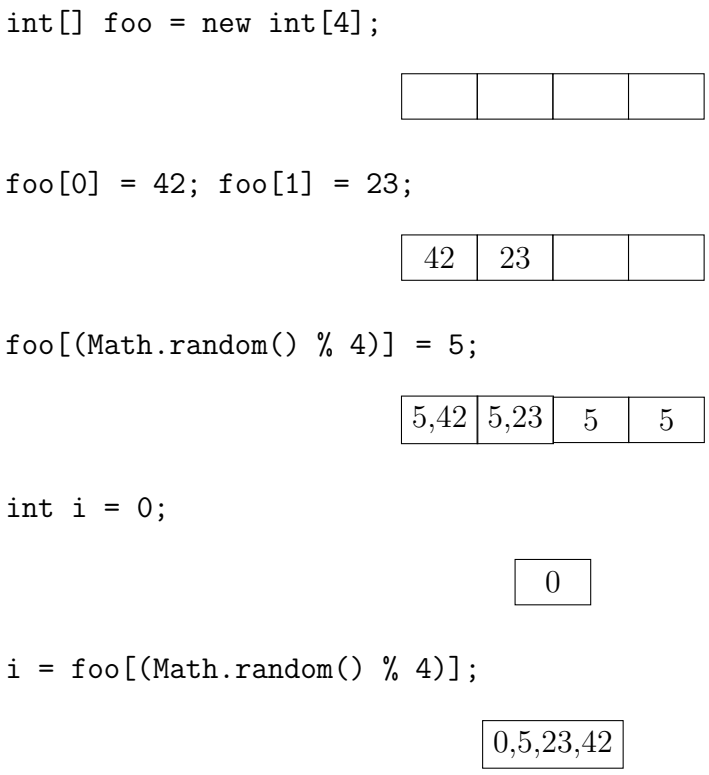

Figure 3.1: Example of array cell usage.

3.1.2 Detecting Aliases

Whenever an array is written, the data flow analysis has to calculate the new cell information and forward the new information to all other instruction nodes,

```
1  private byte foo[] = { 1, 2, 3};
2  private byte bar[] = foo;
```

Listing 3.3: Simple example for array aliasing.

that are using that array. For primitive field types this is simple, as they are read and written explicitly using the fields name in the instruction reading or writing the field. In case of arrays this gets slightly more complicated, as there can be more than one field or local variable referencing the object representing the array, an *alias*. See Listing 3.3 for a very simple example of array aliasing. Whenever *foo* is written, the data flow information of *bar* has to be updated as well, and vice versa. Missing writes to array aliases lowers the precision of the data flow analysis information and could lead to arrays being falsely marked as constant. Furthermore, the compiler can be tricked into omitting an array bounds check, based on an incomplete range of possible values. Keeping track of all array aliases is mandatory.

The extended escaped analysis (EEA) described in Section 2.5 already takes care of the alias analysis, as it keeps track of references passed around by method invocations, return statements and assignments. The following describes how a list of all aliases for an array is extracted from connection graph (CG) built by the EEA.

The array alias analysis searches for array writes and reads in the intermediate code. It retrieves the connection graph node representing the array variable used in the instruction, as the EEA saves a mapping from all variables to their corresponding connection graph nodes. As the variables connection graph node has to be a reference, there needs to be an edge to the corresponding object node. The analysis follows this edge to the object node and inspects the list of nodes pointing to that object node, as these are the nodes that need to be updated after the array is written. Every field reference (fields) and local reference node (stack slots) pointing to the object node will be added to the list of aliases of the variable.

Figure 3.2 shows a connection graph generated by the EEA, for the Java code shown in Listing 3.4. Listing 3.5 shows the emitted C code including the local variables. The *twod* static field is shown as a global reference node, pointing to the same object node as *obj2_0*, which is the stack slot used for reading from *twod*, changing *oned*'s reference. Two array field reference nodes originate from the object node, one for each used array index. *obj2_1* is the stack slot used to write the value 23 to *oned* as well as *twod[0]*.

The array alias analysis would begin with the investigation of the write to *obj1_1*, retrieving the corresponding local reference node in the connection graph, following the edge to phantom node, representing the array it is pointing to. The analysis now searches the list of nodes pointing to the object node, and in that

```
1  public class Example implements Runnable {
2
3          static private byte[][] twod = { {0}, {1} };
4          private byte[] oned = { 23 };
5
6          public void run() {
7                  oned[0] = 42;
8                  oned = twod[0];
9                  oned[0] = 23;
10                 System.exit(twod[0][0]);
11         }
12 }
```

Listing 3.4: A more complex aliasing example.

case adding *oned* to the aliases of *obj1_0*. The next array write is the one writing 23 to *obj2_1*. Again, the analysis follows the edge to the object node, this time adding *twod[0]* and *oned* to the list of aliases.

If the array alias analysis had been omitted, the results of the data flow analysis would declare *twod* a constant array, additionally the content of *twod* would be propagated and the result of the array read instruction, defining the exit status of the program, would have been falsely replaced with a 0, instead of a 23.

3.2 Detecting Runtime-Final Fields

So far, the data flow analysis saves information about array contents and ensures the accuracy of that content by gathering alias information. To be able to mark a field as constant, in addition to constant content, the field's reference needs to be constant as well. When the data flow analysis comes across an instruction reading a non-`final` field, the lattice cell information for that field is computed from all writes to this field, but also a zero value is added. This is done in case the field is read before it has been written, because Java implicitly initializes all variables with a zero value. A `final` field does not require adding that implicit zero value, In the context of field variables, Java's `final` modifier ensures that the field's reference is only set once and has to be set by the end of the constructor, or, in the case of a `static` field, by the end of the class constructor.

A programmer could have forgotten about adding the `final` keyword to a field that was meant to be constant, or it is just not possible to add the keyword because the variable potentially could change its value. In this case, with the help of the reachability information of the data flow analysis, it is possible to detect that all nodes writing to this field, except for the initialization, are unreachable, making the field *runtime final*.

The following three steps explain, when a `static` field can be declared runtime

```
1  void c7_Example_m3_run(object_pointer obj0)
2  {
3      jint i3_0;
4      object_pointer obj1_0;
5      object_pointer obj2_0;
6      object_pointer obj2_1;
7
8      obj1_0 = (ACCFIELD_C7_EXAMPLE_C7F4_ONED(obj0));
9      BYTE_ARRAY_LEA(obj1_0, (0)) = 42;
10
11     obj2_0 = SC7_EXAMPLE_C7F1_TWOD(((domain_t *) &dom1_DDesc));
12     obj2_1 = OBJECT_ARRAY_ALOAD(obj2_0, (0));
13     (ACCFIELD_C7_EXAMPLE_C7F4_ONED(obj0)) = (object_pointer)obj2_1;
14
15     BYTE_ARRAY_LEA(obj2_1, (0)) = 23;
16
17     i3_0 = BYTE_ARRAY_ALOAD(obj2_1, (0));
18     ShutdownOS((StatusType) i3_0);
19 }
```

Listing 3.5: Generated C code of Listing 3.4.

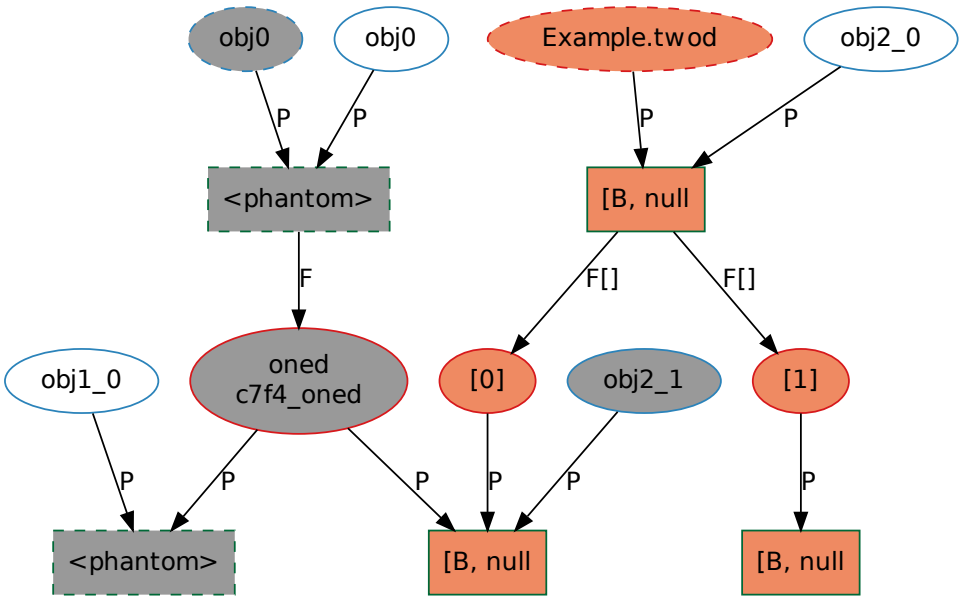

Figure 3.2: Connection graph generated by the extended escape analysis.

final. The following is only sufficient for `static` field members, since the only place to check for the initialization is the class initializer, as opposed to the constructor and all super classes with non-static fields.

Assembling a list of writes: A simple analysis pass scans the intermediate code for a writes of every static field.

Counting and analyzing the writes: Fields written more than once or written outside their class initializer cannot be runtime final. In the case of an array, every index has to be initialized.

Ensure write-before-read: The field has to be initialized before it is being read, so the class initializer method has to be checked accordingly. A basic block in which an instruction node is reading the field or calling a method (which could potentially use the field) has to be dominated by the basic block initializing the field. That means, that all paths in the class initializer leading to the basic block that is reading the field or calling a method, have to go through the basic block writing the field. If there are method invocation instructions or read instructions using the field in the same basic block as the initialization, the write has to happen before them.

If the three requirements are met, the field can be marked as runtime final and a field with constant content can be declared constant after all. Additionally, the Java language specification states that variables of primitive data types marked `final` are constant. The lattice value of a primitive variable, that is only written once and fulfills the requirements of a `final` variable was not constant before this analysis. In addition to its initial value, the data flow analysis always added the implicit zero value. With the runtime final analysis marking this variable runtime final, the implicit zero value can be removed, potentially creating the possibility to convert some conditional branches, marking more code paths unreachable. The dead code could again contain write instructions for other static fields, making even more fields runtime final. Furthermore, runtime final fields of primitive types can, in most cases, be removed, as the byte code instructions fetching their value can be replaced with their constant value.

3.2.1 Emitting Constant Arrays

When the data flow and alias analysis are finished gathering all array information, it is time to find out which arrays are constant, so they can be emitted as constant, pre-initialized structures and to change the intermediate code, replacing the instructions allocating memory for them and writing to them. Arrays will be declared constant, if they are either `final` or at least runtime `final`. So

```
1  /* Typedefs for constant arrays */
2  #include "byte_array.h"
3  typedef struct {
4          ARRAY_HEADER
5          const jbyte data[3];
6  } byte_array3_t;
7
8  /* Constant arrays */
9  const byte_array3_t const_arr1 ALIGN4 ={
10 /*.gcinfo=*/ 1, /*.class_id=*/ BYTE_ARRAY_ID, /*.size=*/ 3,
11 { 1, 2, 3 } /* data */,
12 };
```

Listing 3.6: Constant array `typedef` and `struct` emitted by JINO

a constant array has to be initialized by the end of its (class) constructor and the alias analysis has to prove that no other variable referencing this array has altered the array's content.

A transformation pass was added, running after the optimization loop and before the intermediate code gets translated to C. This transformation pass analyzes array write instructions in the intermediate code, checking if the array that is written to is constant. It does so by checking the data flow cell of the array received by the data flow analysis, as well as the data flow information of the aliases, if there are any. If the array content is constant, the pass removes the array write instruction, as it is redundant and saves the array content in the form of the array cell, so it can be emitted later. If the "defining statement" of the variable referencing the array is an allocation instruction, like in Listing 3.2, this allocation instruction will be saved as well, so it can be replaced later, with a reference to the constant array in the translation process.

The effect of the pass can best be visualized by comparing the initialization of an array like it is shown in Listing 3.2. The transformation removes the write instruction and replaces the allocation instruction, resulting in a much shorter and simpler code:

```
        obj1_0 = const_arr1;
```

Finally, in the translation stage that is generating the C code, JINO will emit a `typedef` for every combination of every different array type and array size for all saved constant arrays. After that it will emit the saved array's contents in form of `const structs`. Listing 3.6 shows an example defining a structure of a constant byte array with the size three. The `const_arr1` structure represents the content of the array from the beginning of this chapter in Listing 3.1.

3.3 Adjusting the Backend and Runtime Environment

Up to this point, JINO is able to detect constant arrays, emit them in form of C `struct`s and substitute allocations with references to this structures. The runtime environment needs adjustments to handle constant data in general. String literals in JINO, although obviously constant, could have been in ROM before the changes described above were developed, but because of the way the garbage collector works, this was not possible. The garbage collector needs to change object header information, in order to work correctly. Certain architectures need special handling of constant data as well, as it resides in memory regions that require other modes of access.

3.3.1 Adjusting the Garbage Collector

If enabled, at some point during the program's execution, KESO's garbage collector will follow object references, starting from a *root set*, checking which objects are still reachable. Objects that are not referenced anywhere, are freed. While following the object references, the garbage collector detects if it has already visited a certain object by marking the object with a certain color. This is done by changing a bit in the field reserved for garbage collector information in the object header. The color which is used to mark already visited objects changes every time the garbage collector is run. That way, objects that have been analyzed in the current run, need to be marked with the current color of the run, by toggling their color bit.

When the garbage collector tries to toggle the color bit of one of the constant arrays, it will likely result in a segmentation fault, if the target's platform features memory protection of some kind. The garbage collector needs some way to identify constant objects, so it knows that it can ignore them. Two different solutions for this problem were developed in this thesis.

Ignore objects by address: One way to identify objects to be ignored by the garbage collector is to put them into a memory region of which the boundaries are known at runtime. This was done by letting JINO's emitted C code be marked with a special ELF section attribute `section(".rodata.keso")`, telling the linker to put the data into that specific section of the binary. This section will be appended to the `.rodata` section, if the linker script shown in Listing 3.7 is added to the regular linker script. The linker script provides symbols for the beginning and the end of the memory region where the constant data lives. These symbols can be used in the C code. The loader resolves them two real addresses at runtime and the garbage collector can check if an object lies between start and end of the KESO specific section and can simply ignore the object in that case.

```
1  SECTIONS
2  {
3          .rodata :
4          {
5                   __keso_glob_start = .;
6                  PROVIDE(__keso_glob_start = .);
7                  *(.rodata.keso)
8                   __keso_glob_end = .;
9                  PROVIDE(__keso_glob_end = .);
10         }
11 }
```

Listing 3.7: Linker script exporting symbols for the KESO specific section.

Ignore objects by color: Another method that is more architecture and tool chain independent, is to make the garbage collector ignore certain objects by reserving a special bit in the garbage collector information byte of the object header, as a marker for constant objects not to be touched. Figure 3.3 shows the layout of the garbage collector information structure. Only two bits of the byte are always in use at the moment. One bit for the color, and the most significant bit is always zero, to distinguish the object header form a regular reference. Additionally to the two existing bits, one bit was be reserved, to mark an object untouchable by the garbage collector. Instead of checking if the object's memory address is within boundaries of read-only memory, the garbage collector checks if the "constant bit" in the object header is set.

3.3.2 Constant Objects on the AVR Platform

The AVR 8-bit microcontroller platform supported by KESO adds an additional challenge to allocating constant data in read-only memory. The AVR's architecture separates instructions and data into two different physical locations. Instructions are stored in the non-volatile flash memory of the microcontroller, whereas data is placed in SRAM. As the available size of RAM on that platform is only a small fraction of the available size of flash memory for code, it is desirable to put constant data into the AVR's program memory. However, since data and instructions are physically separated, the C compiler stores constant data in the SRAM as well, even if they are marked with the `const` modifier. This does not only waste precious RAM space, but makes the constant data modifiable again, as

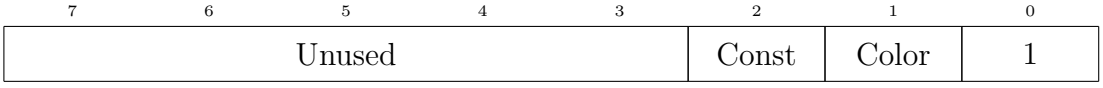

Figure 3.3: Layout of the garbage collector information byte.

there is no memory protection unit to create non-writable sections in the SRAM of the AVR.

Using the AVR's Program Memory

The AVR instruction set provides a way to read data from the program memory and loads it into a register. Reading data from program memory requires the usage of a special assembler instruction so the microcontroller knows that it has to fetch the data from a different memory space. This instruction is called `lpm`, "Load from Program Memory", loading one byte of program memory from an address, stored in a 16-bit address register, into a register.

JINO stores constant data in the AVR's program memory by emitting a special attribute provided by the GNU C compiler, called `PROGMEM`, when defining the variable holding the constant data. This keyword tells the linker to put the variable into program memory. Now that the variable resides in a different memory space, the C code has to be adjusted as well, or else the instruction reading the variable will read from the address in the wrong memory region. This is done by adding a macro, provided by the compiler, around every read access of constant variables marked with the `PROGMEM` attribute. This macro tells the C compiler to insert the special "Load from Program Memory" instruction in the generated assembler code.

Class Storage and Dispatch Table on the AVR

As mentioned in Section 2.2, JINO maintains two arrays containing object-oriented related information, namely the class storage, containing a list of all classes and knowledge about their hierarchy and size, as well as the dispatch table, needed for virtual method calls, in case there is more than one method candidate able to be called. These two arrays will obviously never change, at least not under the premises of KESO, which deny loading of classes at the runtime.

Storing the class storage and the dispatch table in program memory was already implemented in JINO before this thesis. JINO provides a compiler flag emitting a workaround for the AVR architecture, or rather Harvard architectures in general, making it possible for the dispatch and class storage table to reside in the program memory. Instead of generating arrays containing the class and virtual method entries, it emits methods containing `switch` statements, returning the information requested via the method parameters as the result of a method call. This way the information formerly stored in arrays is saved in program memory space, encoded in conditions and return statements.

An alternative to the `switch` encoding was added, using the avr-gcc's `PROGMEM` attribute. As every access to the dispatch table and class storage is done via a macro already, a compiler option was added to mark the dispatch table and

class storage array with the `PROGMEM` attribute and add the "Load from Program Memory" macro to the macro accessing the dispatch table or class storage.

Constant Arrays on the AVR

Now that the class storage and dispatch table can be stored in the AVR's program memory, it would save additional space in the target's SRAM if constant arrays could be saved in program memory as well. Placing the constant arrays into program memory is done by marking them with the `PROGMEM` attribute again. Reading the constant variables placed in the program memory needs special handling in the emitted C code, too. This is handled the same way it as reading from the class storage and dispatch table. The instructions reading from the ROM allocated array are wrapped in a special gcc macro, telling the C compiler to emit the `lpm` instructions, to load the data of that address from program memory, and not from RAM.

Constant Data on the AVR and Garbage Collection

Placing constant arrays and strings in the program memory of the AVR adds one limitation to the possible options of the application's runtime configuration. The garbage collection must not be enabled. Enabling the garbage collector and having some objects living in program memory and some in RAM would be hazardous, because the garbage collector has no way to detect in which memory space the respective object is placed. The garbage collector only sees an address, but the address can be a valid address in both program memory as well as in data memory. With no special precautions taken, this could lead to garbage collector mangling data SRAM, when it tries to toggle the color bit of an object at an address, that actually points to program memory, for example. Since the demand for garbage collection in the area where AVR microcontrollers are used is probably low, this should not be a problem.

3.4 Summary

JINO's data flow analysis was enhanced to support array information, making it possible to detect arrays with constant content. The correctness of the array content is ensured by using the alias information computed by the extended escape analysis. A new analysis pass was added, detecting if a field fulfills the conditions of the `final` modifier, without being declared `final`. Constant arrays are emitted as pre-initialized structures by the backend. Objects referencing these structures are ignored by the garbage collector and can even be placed in ROM on the AVR's Harvard architecture.

4 Evaluation

During this thesis different changes to JINO's runtime environment were made and new compiler passes were added, detecting constant data and generally making it possible to place constant data in the read-only memory of the programs target. The following will now measure how these changes and additions affect the size of the generated binaries and the impact on the programs execution performance.

4.1 Benchmarks

To measure the effect of the changes, there will be data based on the compilation of two different programs. One is CD_x [5], a real-time Java benchmark, emulating an aircraft collision detector. The platform chosen as a benchmark for the AVR related changes is the *Mica2* sensor node, running a traffic light application.

The CD_x benchmarks were performed on x86 hardware and on a TriCore CPU. The AVR code was not run on actual hardware, but in the Avrora AVR simulator. Avrora provides different simulation and profiling methods for AVR code, among them statistics about the amount of cycles the processor spent in each method. The AVR platform has no advanced features like instruction caches, so the amount of cycles each assembly instruction takes, is the exactly the time that is listed in the processors data sheet. See Table 4.1 for details about the test environment.

Benchmark	Hardware	OS	Toolchain
CD_x on-the-go	Infineon TriCore TC1796	CiAO be3999	gcc 4.5.2, binutils 2.20
CD_x simulated	Intel Core2 2.40GHz	Linux 3.4.74	gcc 4.7.2, binutils 2.22
Traffic light	AVR Atmega128 (simulated)	JOSEK	gcc 4.7.2, binutils 2.20.1

Table 4.1: Specification of the test environment.

4.1.1 CD$_x$

The CD$_x$ implements an aircraft collision detector, calculating the distance between aircrafts from radar frames generated by an air traffic simulator.

There are two operation variants of the CD$_x$. The *simulated* CD$_x$ runs two different threads, one for the collision detector and one the air traffic generator generating radar frames and passing it to the collision detector. The *on-the-go* variant generates the radar frames for the collision detector in the same task, with a simpler radar frame generation method. As the *on-the-go* variant is of a much simpler structure it is the only CD$_x$ variant that was tested on the TriCore. The *simulated* multi-domain version does not fit into the TriCore's memory and was only compiled for x86 to measure the changes of the binary size.

4.1.2 I4Copter

The I4Copter [9] is quadrotor helicopter, used as a research platform for safety-critical embedded software. It runs on top of the CiAO operating system [7], with different software modules controlling the copter's rotors based on the input by gyroscopic sensors, accelerometers and the remote control. Different software modules, for example the module managing the SPI bus controller, are available as KESO ports and were added to the list of benchmarks, as an example for a real world Java project.

4.1.3 AVR Traffic Light

The AVR test application simulates a traffic light, displaying its current state with three LEDs of red, yellow, and green color. The default state of the traffic light is the red light and a state change can be triggered by sending a command via a serial connection. There are two tasks in the application. The remote control task, waiting for the signal switch command, triggering the actual traffic light task, which then cycles through the colors of a regular traffic light, displaying the respective color for a pre-defined amount of time, until it reaches the red-light state again.

4.2 Measurements

Two different kinds of data will be measured in the following, using the setup described above: the effect of the optimizations regarding the generated binary size, especially the differences between text and data section, as well as the impact on the execution time of the compiled application.

```
1 if (v[i].equals("MAX_FRAMES")) {
2   Constants.MAX_FRAMES = Integer.parseInt(v[i + 1]);
```

Listing 4.1: Static field members set by parameters in the CD_x.

4.2.1 Binary Size

The data was gathered using the `size` utility from the GNU binutils package. Both variants of the CD_x were analyzed for changes in the binary size and were compiled with the x86 as target architecture and Trampoline OSEK implementation as the operating system, because the target architecture and OS are of no real relevance measuring the binary size and the measurements could be automated in a more convenient way.

Runtime Final Analysis

The first thing measured is the impact of the runtime final analysis. This analysis detects static field variables that are not marked `final`, but meet the conditions of the keyword after all.

Table 4.2 shows the difference in binary size created by the runtime final analysis pass. As the analysis did not find any runtime final variables in the traffic light application, as well as in the I4Copter code, it had no effect on the binary size and therefore no results for these applications are listed in the table. However the analysis found several runtime final variables in the CD_x. The on-the-go variant has 71 runtime final fields in total. In the simulated variant 75 runtime final variables were found. The CD_x benefits a great deal from the runtime final analysis, because of a class literally called `Constants`, containing different static field members, most of them `final` integers, controlling parameters of the air traffic generator. Under normal circumstances the CD_x can be configured with flags passed to the binary on execution, changing some of the constants in the Constants class, therefore some of them cannot be marked `final`. Every KESO application is statically configured, that way there is no need for passing further configuration parameters. Code like in Listing 4.1, containing a condition checking if a certain parameter was given to the binary and setting the respective constant accordingly, is marked dead. The analysis detects 30 runtime final fields in the Constants class and all of them are removed by the optimization pass removing fields that are never written nor read, as all instructions reading the field's value can be replaced with the actual value of those fields. Enabling the runtime final analysis removes 35 additional fields in the unneeded-field-removal pass in the simulated variant. In the on-the-go variant 33 more fields are removed.

Benchmark	Text	Data
CD_x simulated, runtime final analysis disabled	275,986	47,680
CD_x simulated, runtime final analysis enabled	262,218	45,608
	-5%	**-4%**
CD_x on-the-go, runtime final analysis disabled	86,166	3,820
CD_x on-the-go, runtime final analysis enabled	74,474	2,240
	-14%	**-41%**

Table 4.2: Result of the runtime final analysis. Segment sizes in bytes. The runtime final analysis renders 4 to 14% of previous code dead and reduces the data segment usage of the simulated variant by 4%, with the on-the-go variant even by 41%.

Constant Data in ROM

The primary focus of this thesis is the amount of ROM allocatable data. How much of the targets memory can be freed by moving constant data from the data section to read-only memory. Table 4.3 displays the shifting of constant data from the data to the text section. Both CD_x benefit from the fact that strings can now be placed in the ROM section, which was prohibited by the garbage collector. Besides strings, no further constant data was detected in the CD_x on-the-go. The simulated variant contains three additional constant arrays in parser related code, but they only contain one element, which is a reference to a string.

The I4Copter modules did not contain any constant arrays, nor strings. However the traffic light contains strings that are now placed in program memory, as well as the table containing the information about the duration of each color phase and the next color to be shown. This state table is detected as a constant array and is emitted as a fully initialized C structure and placed in program memory, too. Adding to the shift of bytes from data to text section in the traffic light binary, is the fact that the virtual method table and the class storage array are no longer occupying SRAM space but are placed in flash memory.

4.2.2 Execution Time

To measure the impact on the execution time the CD_x on-the-go variant was executed on the TriCore board with the specifications listed in Table 4.1. The data that was compared is output generated by the CD_x benchmark. The CD_x measures the timespan for the calculation of the collision probability, based on incoming radar frames. The CD_x prints these times to the standard output. 50 iterations were deemed adequate for this thesis. The AVR code was analyzed with the help of the Avrora AVR simulator, as it offers different profiling options and cycle-accurate simulation. Changes to the performance of the I4Copter were

Benchmark	Text	Data
CD_x simulated, without ROM allocation	262,218	45,608
CD_x simulated, with ROM allocation, excl. strings	262,282	45,560
	+64	**-48**
CD_x simulated, without ROM allocation	262,218	45,608
CD_x simulated, with ROM allocation	306,154	1,684
	+43,936	**-43,924**
CD_x on-the-go, without ROM allocation	74,474	2,240
CD_x on-the-go, with ROM allocation, excl. strings	74,474	2,240
	+0	**-0**
CD_x on-the-go, without ROM allocation	74,474	2,240
CD_x on-the-go, with ROM allocation	75,658	1,052
	+1,184	**-1,188**
Traffic light, without ROM allocation	8,626	940
Traffic light, with ROM allocation (excl. strings)	8,738	828
	+112	**-112**
Traffic light, without ROM allocation	8,626	940
Traffic light, with ROM allocation	8,890	672
	+264	**-268**

Table 4.3: Results of ROM allocation. Segment sizes in bytes. Strings make up the biggest part of the amount of data shifted from the data to the text segment.

Benchmark	NPC emitted	NPC removed
CD_x, on the go, RFA disabled	204	796
CD_x, on the go, RFA enabled	142	621

Table 4.4: Number of null-pointer checks emitted and number of null-pointer checks removed, as JINO's data flow analysis guarantees that these objects are valid. The number of null pointer checks is reduced by 30%, the overall number of removed checks went down by 22%, attributed to the general code size reduction.

not analyzed, as none of the optimizations developed in this thesis yielded any results.

Runtime Final Analysis

Figure 4.1 shows the measured time of a CD_x, compiled with the runtime final analysis (RFA) enabled, relative to the time measured with an instance compiled without the analysis. Again, the AVR traffic light is not listed in the results, as the runtime final yields no changes on that application. The runtime final analysis gains an overall execution time improvement, with a mean value of 10%. This is mostly due to the folding of conditional statements, removing the dead basic blocks and removed `null`-pointer checks. Table 4.4 shows the difference in the number of emitted null pointer checks, which is reduced by 30%. The number of null checks that JINO removed, as the the information of the data flow analysis proved that the object in question is always valid, was reduced as well. This is attributed to general decrease of code size.

As the runtime final analysis not only declares fields of primitive data types, like integers, as runtime final, but object references as well, this gains performance improvements, too. Some static fields used in the CD_x code and in the library code originate from the usage of the *singleton* design pattern. With this pattern, there is only one instance of a class, which can be received by a static method provided by that class. See Listing 4.2 for an example used in the CD_x. The singleton field is only written once and guaranteed to be non-`null`, so all null checks regarding that particular field can be removed.

ROM Allocation

In Figure 4.2 two relative time measurements are shown for two CD_x instances that were compiled with ROM allocation enabled. One was built with the garbage collector ignoring constant objects by checking if the object's address lies in the special memory range for constant objects, provided by the linker script. The second instance was built with the garbage collector ignoring constant objects of

```
1  public class Clock {
2
3    static Clock singleton = new Clock();
4
5    public static Clock getRealtimeClock() {
6      return singleton;
7    }
8    public AbsoluteTime getTime() {
9      long nanos = System.nanoTime();
10     return new AbsoluteTime(nanos / 1000000L, \
11                             (int) (nanos % 1000000L));
12   }
13 }
```

Listing 4.2: Example of the singleton design pattern. The instance of the class can only be acquired via the `getRealtimeClock()` method.

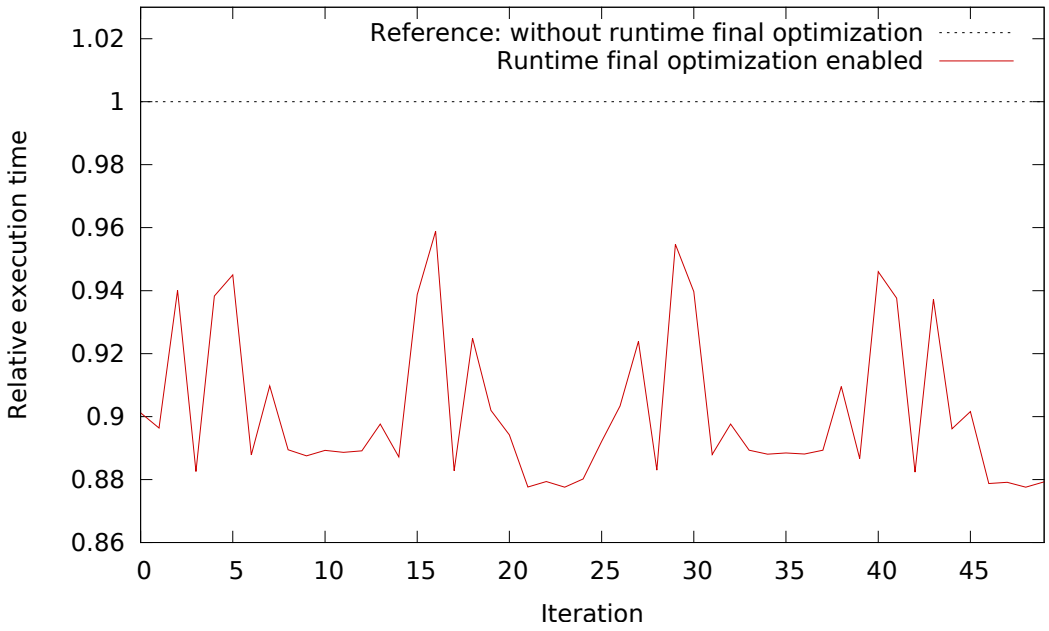

Figure 4.1: Relative execution time of the CD_x on-the-go with runtime final analysis. Execution time improved by 5 to 12%.

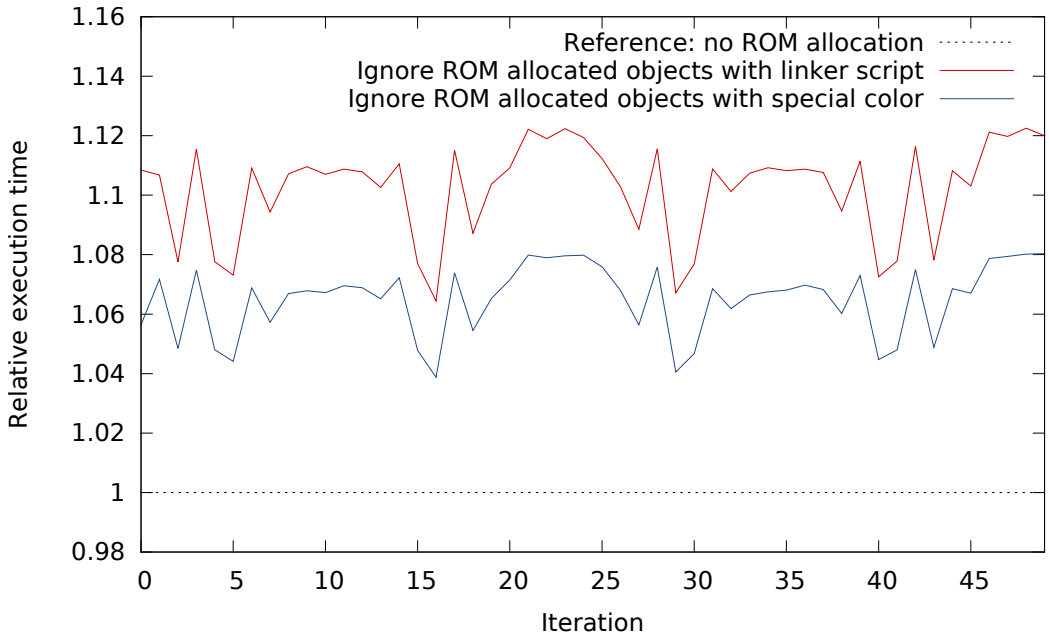

Figure 4.2: Relative execution time of the CD_x on-the-go with enabled ROM allocation. Ignoring constant objects in the GC via memory range check induces 8 to 12% penalty on the execution time, whereas checking for a special color in the object header's `gcinfo` byte only decreases performance between 4 and 8%.

a certain color, represented by a special bit in the object's header. The results show that the method ignoring by color has less negative impact on the execution time, with a mean of 6% slowdown, compared to 10% of runtime penalty due to the linker script memory range method.

To measure changes in the performance due to ROM allocation in the traffic light application, the profiling feature of the Avrora simulator was used. The profiler measures the amount of time it spent in each function. The time is measured in cycles. To measure the performance, the traffic light application was run with profiler enabled, measuring the time spent in each function during one full cycle from red light to green and back to red light again. The method that was compared is the *setLights* method, responsible for enabling and disabling specific LEDs and setting the alarm trigger for the display of the next color. This method almost exclusively works with the array containing the states and the amount of seconds that each color is displayed, so the impact on the execution time when storing this array in program memory is best revealed here. Table 4.5 displays the result of the Avrora cycle count of the `setLights` method, showing a 24% runtime overhead with the ROM allocated state array. The Atmega128 manual [1]

Function	ROM allocation	No ROM allocation
TrafficLightTask `setLights`	877	671
TrafficLightTask `clinit`	31	115

Table 4.5: Time spent in the method responsible for toggling the LEDs and the class initializer of the AVR traffic light application, measured in cycles.

Inst. (no ROM alloc.)	*Cycles*	**Inst. (ROM alloc.)**	*Cycles*
`movw r30, r16`	1	`movw r30, r28`	1
`ldd r22, Z+5`	2	`adiw r30, 0x05`	2
`mov r28, r17`	1	`lpm r28, Z+`	3
`std Y+2, r22`	2	`movw r30, r16`	1
		`std Z+2, r28`	2
	6		9

Table 4.6: Comparison of the emitted assembly instructions and their cycle count, reading a byte from program memory versus reading a byte from RAM.

states that the `lpm` instruction responsible for reading data from program memory takes three cycles to complete. Table 4.6 shows the assembly instructions emitted and the cycle count for reading a value from an array saved in RAM with `ldd` (load indirect from data space), versus reading from program memory with `lpm`. Additionally the cycle count for the class initializer of the `TrafficLight` class was measured. The cycle count is down by 73% with ROM allocation enabled, because the state array does not need to be allocated and initialized, only the reference to it has to be assigned to the static field variable pointing to it. By removing the allocation of the state array, the method responsible for allocating objects on the heap is not required anymore, as every other resource or variable already has space reserved in the heap or is stack allocatable, so ultimately no garbage collector code would be needed for this application.

4.3 Summary

The evaluation of the runtime final analysis and the ROM allocation of constant data show that the effect of the optimizations that were developed during this thesis are very application specific. Neither the runtime final analysis, nor the constant arrays detection had any effect on the I4Copter application, since there are no runtime final variables or constant arrays. The runtime final analysis, however, did affect the CD_x execution performance and binary size in very positive way, removing 30% of the null pointer checks and adding a 10% performance

gain. Also the ROM allocation shifts great amounts of bytes from the data to the text section of the CD_x, although most of that data consists of strings and no real world embedded-system application would contain that large amount of strings. Additionally reading data from ROM slows down the execution time of the application.

The traffic light application gains nothing from the runtime final analysis. However, the ROM allocation of the array containing the state table responsible for the correct timing and display of the correct LED has different positive implications on the generated code. It slows down the application, because the instruction reading from program memory takes more time to complete than a load from RAM, but it reduces the amount of required memory and in the case of this specific application removes the need for a garbage collector all together, reducing the requirements of program memory size. Especially with the AVR as target, ROM allocation is a space-time tradeoff, but environments deploying this type of microprocessor are probably not aiming for high performance anyway.

5 Conclusion

In the scope of this thesis, different optimization passes were added to the KESO Java Virtual Machine, which is designed for statically configured systems in the embedded software area, compiling Java to C code ahead of time. The optimizations detect constant data in application bytecode, in the form of arrays and primitive fields. The JVM backend was enhanced to make it possible to place this newly detected data in read-only memory, as well as already existing constant data, like string literals and runtime data structures. The necessary measures to support Harvard architectures, like the AVR, providing separate locations for program memory and data, were taken as well. Finally, the runtime environment was changed, to protect the constant objects from unwanted attention from the garbage collector.

Constant arrays were detected by adding the support for this data structure to the already existing data flow analysis. Possible aliasing, where more than one variable is able to alter the content of an array, was detected, too. This was done by extracting information from another existing analysis pass, originally used for escape analysis.

Additionally, objects and primitive fields that never change their initial reference or value during runtime can be declared runtime final, if they fulfill the requirements of Java's `final` modifier, but are not declared as such.

The garbage collector was taught to ignore constant objects, either by ignoring a certain memory range, reserved for constant objects, or by setting a special bit in the object's header. For the AVR, the backend emits special macros, to tell the assembler to generate appropriate instructions when it tries to read values from constants placed in the processors flash memory.

The effects of the developed optimizations on the execution time and binary size of different applications were measured. The results show that the impact of the ROM allocation and runtime final analysis are highly application depended. If the application provides exploitable data, like a lookup table, or strings, the optimizations moves this data away from RAM to the read-only data section, although at the expense of execution performance. Storing data in the program memory on the AVR is a space-time tradeoff, too. The runtime final analysis can improve the performance and decrease the binary size under the right circumstances.

Bibliography

[1] Atmel Corporation. *ATmega128 8-bit AVR Microcontroller with 128KBytes In-System Programmable Flash*, June 2011. Rev. 2467X–AVR–06/11.

[2] AUTOSAR. Specification of operating system (version 5.0.0). Technical report, Automotive Open System Architecture GbR, November 2011.

[3] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, November 2003.

[4] Christoph Erhardt. *A Control-Flow-Sensitive Analysis and Optimization Framework for the KESO Multi-JVM.* Diplomarbeit, Friedrich-Alexander University Erlangen-Nuremberg, March 2011.

[5] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CD_x: a family of real-time java benchmarks. In *JTRES '09: 7th Int. W'shop on Java Technologies for real-time & embedded Systems*, pages 41–50, New York, NY, USA, 2009. ACM.

[6] Clemens Lang. *Improved Stack Allocation Using Escape Analysis in the KESO Multi-JVM.* Bachelorarbeit, Friedrich-Alexander University Erlangen-Nuremberg, October 2012.

[7] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX ATC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[8] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2011-08-17.

[9] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. I4copter: An adaptable and modular quadrotor platform. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 380–386, New York, NY, USA, 2011. ACM.

[10] Christian Wawersich. *KESO: Konstruktiver Speicherschutz für Eingebettete Systeme*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2009.

[11] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13:181–210, April 1991.